# Towards Business Processes Orchestrating the Physical Enterprise with Wireless Sensor Networks

Fabio Casati[‡], Florian Daniel[‡], Guenadi Dantchev[†], Joakim Eriksson[*], Niclas Finne[*], Stamatis Karnouskos[†], Patricio Moreno Montero[**], Luca Mottola[*], Felix Jonathan Oppermann[+], Gian Pietro Picco[‡], Antonio Quartulli[‡], Kay Römer[+], Patrik Spiess[†], Stefano Tranquillini[‡], Thiemo Voigt[*]
[**]*Acciona Infraestructuras S.A. (Spain),* [†]*SAP AG (Germany),* [*]*Swedish Institute of Computer Science,*
[+]*University of Lübeck (Germany),* [‡]*University of Trento (Italy)*

***Abstract*—The industrial adoption of wireless sensor networks (WSNs) is hampered by two main factors. First, there is a lack of integration of WSNs with business process modeling languages and back-ends. Second, programming WSNs is still challenging as it is mainly performed at the operating system level. To this end, we provide `make`*Sense*: a unified programming framework and a compilation chain that, from high-level business process specifications, generates code ready for deployment on WSN nodes.**

## I. Introduction

Wireless Sensor Networks (WSNs) are small, untethered computing devices equipped with sensors and actuators. WSNs can be easily deployed and are able to self-organize to achieve application goals. Research has made significant progress in solving WSN-specific challenges such as energy-efficient communication. Industry, however, is reluctant to adopt WSNs. We believe this is due to two unsolved issues, integration and unification, schematically shown in Figure 1.

**Integration** refers to the need for strong cooperation of business back-ends with WSNs. Current approaches typically consider the WSN as a stand-alone system. As such, the integration between the WSN and the back-end infrastructure of business processes is left to application developers. Unfortunately, such an integration requires considerable effort and significant expertise spanning from traditional information systems down to low-level system details of WSN devices. Moreover, these two sets of technologies satisfy very different goals, making the integration even harder. This paper presents a holistic approach where application developers "think" at the high abstraction level of business processes, but the constructs they use are effectively implemented in the challenging reality of WSNs.

**Unification** refers to the need for a single, comprehensive programming framework. It is notoriously difficult to realize WSN applications. They are often developed atop the operating system, forcing the programmer away from the application logic and into low-level details. Many programming abstractions exist [1], but are hard to use since they typically focus on one specific problem. To drastically simplify WSN programming, particularly for business scenarios, we need a broader approach enabling developers to use several abstractions at once. In this paper, we present a unified comprehensive programming framework into which existing WSN programming abstractions can blend smoothly.

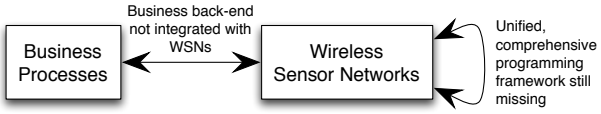


Figure 1. Open problems for using WSNs in business processes.

## II. Application Scenarios

A paradigmatic example of our target scenarios is ventilation in buildings. Fans are commonly operated at a fixed rate, independent of room occupation, resulting in unnecessary ventilation of unoccupied rooms and over-ventilation of sparsely occupied ones, ultimately wasting energy. A smarter strategy may consider room occupation, resulting in sustainable building management. Consider an office environment, in which employees book meeting rooms on the Web through a back-end process notifying the expected participants. Room ventilation is minimal when no meeting is scheduled. Sensors and actuators driven by the business process increase ventilation before the meeting and until either human presence is detected or $CO_2$ levels are above a certain threshold.

Realizing this system requires a tight integration between the business process and the network of sensors and actuators dispersed in the environment, as the application logic needs to extend to the latter. Moreover, implementing the processing for adaptive ventilation complicates application development, as it departs from the traditional data collection—most common in WSN applications—to encompass possibly distributed control loops. Similar requirements are shared by numerous application domains such as predictive maintenance aboard cargo vessels.

## III. Approach

Our design revolves around three fundamental goals:

- `make`*Sense* must *seamlessly integrate* with existing business process technology, providing an adoption path that complements, instead of disrupts, existing methodologies and technologies with WSN ones.

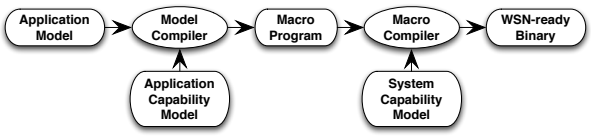


Figure 2. Compiling business process models into WSN-executable code.

- `make`***Sense*** must be *modular* and *extensible*. As we aim for our system to be useful across several real-world applications, extensibility is key to ensure that the programming abstractions and their implementation can be easily adapted to the specificity of the target domain as well as to unforeseen needs.
- for extensibility not to be detrimental to performance, `make`***Sense*** must *self-optimize* w.r.t. high-level performance goals. This is necessary to support long-lasting business processes subject to the randomness of the real world and rapidly changing requirements.

### A. Architecture Overview

Our architecture is based on the separation of concerns provided by a distinction in layers of functionality: *i)* an *application* layer concerned with business processes and their modeling; *ii)* a *macroprogramming* layer concerned with the distributed execution of activities within the WSN; *iii)* a *run-time* layer concerned with the low-level aspects supporting the above and enabling self-optimization. The term "macro-programming" [1] refers to approaches that, unlike node-centric ones, allow specifying the behavior of multiple WSN nodes at once.

A model-driven approach connects the three layers (Figure 2). The application model represents a holistic, network-agnostic view of the entire business process, i.e., including the WSN and the process back-end. It includes performance requirements (e.g., a certain level of reliability, or a minimum lifetime). Details are further described in Section III-B.

Two compilation steps link the layers above. The model compiler takes as input the application model and an application capability model. The latter is a coarse-grained description of the WSN, providing information such as the type of sensors/actuators available and their operations. The model compiler translates these descriptions into a program written in a macro-programming language, described in Section III-C, serving as an intermediate language closer to the reality of WSN systems, yet high-level enough to be potentially used directly by a developer.

The macro compiler takes as input the macro-program generated by the model compiler and a system capability model. The latter provides finer-grained information on the deployment environment (e.g., how many sensors of a given type are deployed at a location). The macro-compiler generates executable code that relies only on the basic functionality provided by the run-time support available on the target nodes. By leveraging the system capability model, the macro compiler can generate different code for different nodes, based on their application role. The executable code contains the mechanisms enabling self-optimization, described in Section III-D.

### B. Business Process Modeling

When integrating WSNs with business processes, most research projects and productive set-ups merely add a service facade to the WSN and orchestrate its services centrally. If middleware is deployed, that is done either purely in a central system [2] or with additional local components close to the WSN or on its gateway [3]. In `make`***Sense***, we use a more radical approach. As our goal is to decrease the effort of programming WSN applications, tools for process modeling are used to create the application top-most level. A process modeler models hybrid processes, of which one part is executed conventionally in a central execution engine, while another part is executed directly by the WSN.

We use and extend the *Business Process Modeling Notation (BPMN)*. By introducing new attributes, the modeler can specify a new *intra-WSN participant*, containing the logic executed by the WSN. As the latter is resource-constrained, we allow only a subset of BPMN elements. Furthermore, we introduced a new *WSN activity* type. This can be used only within the *intra-WSN participant* and is (except for the message activity) the only allowed activity type there. The WSN activity is backed by a meta-model, described in the next section. As WSNs are inherently distributed systems, we also introduced a *Target* attribute for lanes and activities within the *intra-WSN participant*, that allows specifying where the respective logic should be executed, based on labels that are relevant at the modeling layer. Finally, we added performance annotations, expressing that the WSN should optimize its operation for a specific goal (e.g., system lifetime or reliability) within a certain subsets of activities.

To assist the process modeler in creating correct, executable models, we use a set of meta-models that describe the WSN in terms of the logical functionality it provides, along with the way it is embedded into the physical set-up (e.g., which sensing or actuation is supported at which logical location). Instances of these meta-models can be created either manually or through dynamic service discovery.

At run-time, the BPMN process is executed in a distributed fashion. For message exchange between the intra-WSN participant and the other participants, the run-time uses a lightweight protocol, reducing encoded message size by using message structure information on both sides. Communication endpoints caring for serialization and deserialization of messages and for process instance correlation are generated automatically as part of the compilation process.

### C. The `make`***Sense*** Macroprogramming Language

Our intent is not to propose another macro-programming language. Rather, it is to provide a framework where the abstractions contributing to the language are decoupled,

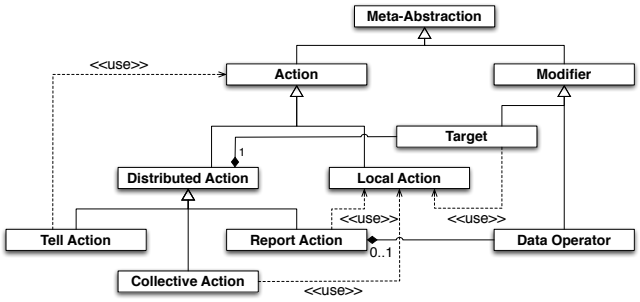


Figure 3. A model for the meta-abstractions of the `make`*Sense* macro-programming language.

leverage on existing implementations, and can be changed or extended easily to suit specific application needs.

This goal influenced the entire language design. To properly identify the units of functionality, reuse, and extensions we defined the notion of *meta-abstraction*, implemented through different "concrete" abstractions, as described later. Abstractions provide the key concepts enabling interaction with the WSN. However, their composition can be achieved by using common control flow statements, provided by a core language that serves as the "glue" among macro-programming abstractions. The core language, in our case a stripped-down version of Java we tailored for WSNs, is also the *trait d'union* between the macroprogramming abstractions and the BPMN business process model.

Figure 3 shows a UML meta-model for the meta-abstractions provided by the macroprogramming language. It focuses on the notion of *action*, a task executed by one or more WSN nodes. Actions are separated into *local*, whose effect is limited to the node where the action is invoked (e.g., acquiring a reading from the on-board temperature sensor), and *distributed*, whose effect instead spans multiple nodes.

Distributed actions are further divided into *tell*, *report*, and *collective* actions. The former two represent the one-to-many and many-to-one interaction patterns commonly used in WSNs to enable communication between the node (the "one") issuing the action and a set of nodes (the "many") where the latter is executed. A tell action enables a node to request the execution of a set of actions on other nodes, e.g., to issue actuation commands or to trigger reconfiguration of system parameters such as the sampling rate. A report action enables a node to gather data from other nodes. Event-based abstractions and periodic, continuous queries both fall in this category. Data acquisition occurring on each target node is specified by a local action given as input to the report action. The output of the local action is returned to the report one. Collective actions, in contrast to tell and report ones, do *not* focus on a special node where the action starts or ends. They enable a global network behavior and are executed cooperatively by the entire WSN through many-to-many communication. An example are distributed assertions [4], where programmers specify a (global) property monitored collectively by the WSN nodes.

Distributed actions may optionally have *modifiers* associated with them, "customizing" their behavior. We defined two modifiers, target and data operator. In our scenarios the nodes possibly differ along several dimensions, both physical and logical. For example, ventilation in Section II requires both $CO_2$ and presence sensors. Programmers must be able to map actions to the set of nodes of interest. A *target* identifies a set of nodes satisfying application constraints, and gives the ability to apply a distributed action to the nodes in this set. Instead, a report action may have a *data operator*, specifying processing performed on the results after gathering and before they are returned to the caller, e.g., to filter or aggregate the data.

To create an instance of a meta-abstraction, a class implementing its interface must be defined in the core language. As abstraction implementations typically closely interact with the operating system, methods of abstraction classes are implemented in C using a native code interface provided by the core language. Some abstractions require extensive configuration, for example, a target needs to define a set of nodes based on their properties [5]. To simplify such configuration, the core language supports the concept of *embedded languages*, code snippets formulated in the declarative configuration language provided by an abstraction. These are efficiently compiled by appropriate compiler plugins, instead of being interpreted at runtime.

### D. Run-Time System

Besides providing a foundation for the distributed protocols in support of the macro-programming language, the run-time system offers self-optimization functionality to adapt the system behavior to changing requirements based on developer-provided high-level performance goals. For example, in the scenario of Section II, the high data reliability required to accurately monitor the persons' presence will correspond to different protocol settings compared to situations with no ongoing meetings, when energy preservation is the major performance concern.

To achieve this functionality, we gather run-time information from the WSN (e.g., network topology and protocol performance) and feed these to a reinforcement learning algorithm that uses simulations to explores the space of possible protocol configurations. At the end of each simulation round, the learning process evaluates the performance obtained with a given protocol setting w.r.t. the application's performance goals. Based on this, we derive self-optimization policies that specify which protocol parameters provide better performance as a function of the current application performance goal. We distribute the policies back to the deployed network where nodes will apply them upon recognizing changes in the current performance goal.

This approach sharply differentiates from existing solutions. Rather than requiring detailed modeling of the individual protocols, we treat the entire application as a

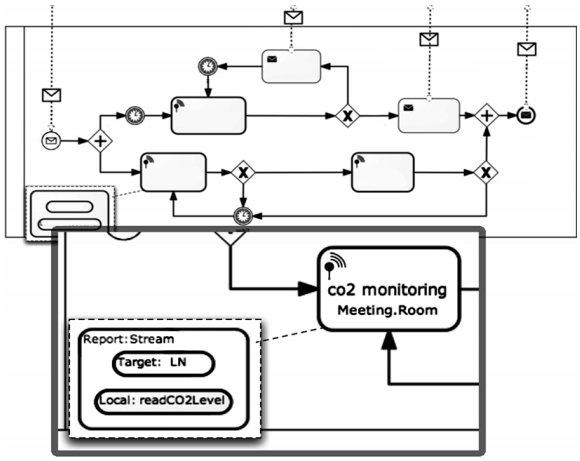


Figure 4. BPMN diagram for a fragment of the ventilation scenario.

black-box. This may lead to sub-optimal solutions, but also enjoys greater flexibility as it lets users add programming abstractions to the framework along with their supporting protocols and have the latter "implicitly" optimized.

## IV. Current Status

We implemented the extended BPMN meta model in *Signavio Core Components*, an open source, browser-based BPMN editor. Our prototype implementation is focused on the model-to-macrocode transformation. Future work includes extending a BPMN runtime with the lightweight messaging protocol in JSON. The macro-compiler prototype is implemented as a multi-pass compiler employing the ANTLR parser generator and the StringTemplate engine. Currently, the compiler is primarily optimized for maintainability and extensibility. We also implemented concrete abstractions for *report* and *tell* actions, and for the *target* modifier. The former is a variation of a standard WSN data collection protocol, whereas the others rely on Logical Neighborhoods [5]. The self-optimization functionality is a separate stand-alone prototype written in Java that we are currently integrating into the `make`***Sense*** run-time.

## V. Case Study

Figure 4 depicts a fragment of the business process model for ventilation we briefly outlined in Section II. The whole process is modeled with two participants, the WSN-aware participant on top and the intra-WSN participant (modeled in more detail) that is converted into an application by generating macrocode. The zoomed part of the process shows a WSN activity that sets up and executes a periodic reading of $CO_2$ sensors in a certain room. By graphically combining abstractions—here a *target* specifying the room and a *local action* to read the sensor are used with a *report* action to collect sensor data—along with meta-information of the current WSN setup, the model becomes rich enough to be transformed into macrocode.

The corresponding code in Figure 5 describes the instructions to define a *target* including all $CO_2$ sensors and to

```
...
code nhoodTemplateS = {:
  neighborhood template CO2Sensors()
    f.getFunction() = "sensor" and t.getType() = "co2"
  create neighborhood co2Sensors from CO2Sensors () :};
Target co2Sensors = lnew LN(sensorNeighborhoodDef);

Report co2Stream = lnew Stream();
co2stream.setTarget(co2Sensors);
co2Stream.setParameter("period", 5 * 60);
co2Stream.execute();
...
```

Figure 5. Macro-programming language fragment for Figure 4.

collect periodic data from them using an instance of *report* action implemented with `Stream`. The abstraction-specific code inside the `code` variable is the Logical Neighborhood [5] custom language. This is used to create an instance of *target*, referring to local actions to retrieve the function and type of node to possibly include in the target. The *target* is given as parameter to a `setTarget` method invoked on an instance of *report*. The remaining method invocations are used to set parameters for the functioning of the `Stream` instance, e.g., its reporting period.

The BPMN model also contains performance annotations. Based on this and monitoring data, the self-optimization functionality tunes the protocols' parameters, e.g., by going into a very low power mode when no meeting is scheduled and no presence of people has been detected.

## VI. Conclusion

We presented early results of the `make`***Sense*** project, which tackles the unification of existing WSN programming abstractions and the integration of WSNs with business process models and back-ends. These issues are hampering industrial WSN adoption, thus, we believe that `make`***Sense*** will foster adoption of WSNs in industry applications.

## Acknowledgments

This work is supported by the European Commission through the `make`***Sense*** and CONET projects.

## References

[1] L. Mottola and G. Picco, "Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art," *ACM Computing Surveys*, vol. 43, no. 3, 2011.

[2] C. Decker, T. Riedel, M. Beigl, L. de Souza, P. Spiess, J. Muller, and S. Haller, "Collaborative business items," in *IET Int. Conf. on Intelligent Environments*, 2007.

[3] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the SOA-based Internet of Things: Discovery, query, selection, and on-demand provisioning of Web services," *IEEE Trans. on Service Computing*, vol. 3, no. 3, 2010.

[4] K. Römer and J. Ma, "PDA: Passive distributed assertions for sensor networks," in *Proc. of the Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2009.

[5] L. Mottola and G. Picco, "Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks," in *Proc. of the Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, 2006.